UM-SJTU JOINT INSTITUTE

Data Structures and Algorithms
(VE281)

Project Report

Project 1

Name: Haorong Lu ID: 518370910194

Date: 1 November 2020

1 Test Setup

In this test, we measure the performance of bubble sort, insertion sort, selection sort, merge sort, extra space quick sort, in-place quick sort as well as the provided std::sort().

The test is based on Ubuntu 20.04 LTS, gcc 9.3.0 with $16 \times 2900MHz$ CPUs. For simplicity, I use Google Benchmark to do the test so that I can get nanosecond level time and the average execution time for several cases easily. The command I use to compile and execute is

```
g++ main.cpp -std=c++11 -isystem benchmark/include -Lbenchmark/build/src
    ↪  -lbenchmark -lpthread -o main -Ofast
./main --benchmark_out=test.csv --benchmark_out_format=csv
```

−Ofast is turned on because if I do not apply the compiler optimization, the std::sort() will always be slower than my quick_sort_inplace(), which is unreasonable, so I enable −Ofast to optimize the performance of STL functions.

2 Result

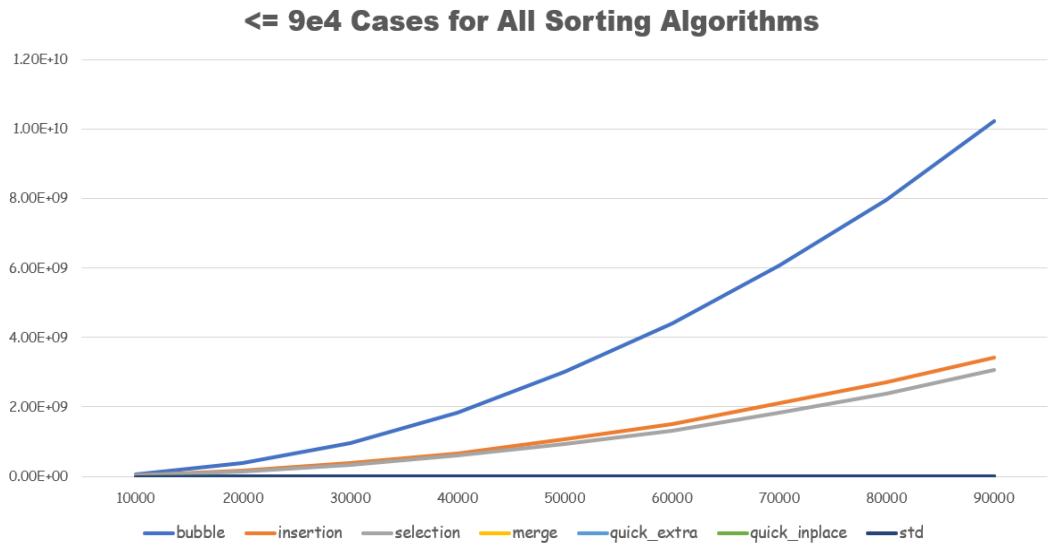

Figure 1: $<= 9 \times 10^4$ Cases for All Sorting Algorithms.

First we plot the performance of all sorting algorithms when the size of input arrays is in the interval $[1 \times 10^4, 9 \times 10^4]$, from the graph above, we can observe that when the input size becomes larger, the execution times of bubble sort, insertion sort, selection sort increase rapidly, and bubble sort has a bigger constant, while the other four sorting algorithms do not show significant growth. We can confirm that the time complexity of bubble sort, insertion sort, selection sort is $O(n^2)$, and merge sort, quick sort have $O(nlogn)$ time complexity.

To show the difference between four $O(nlogn)$ sorting algorithms, we plot the performance of them when the size of input arrays is in the interval $[5 \times 10^4, 1 \times 10^6]$,

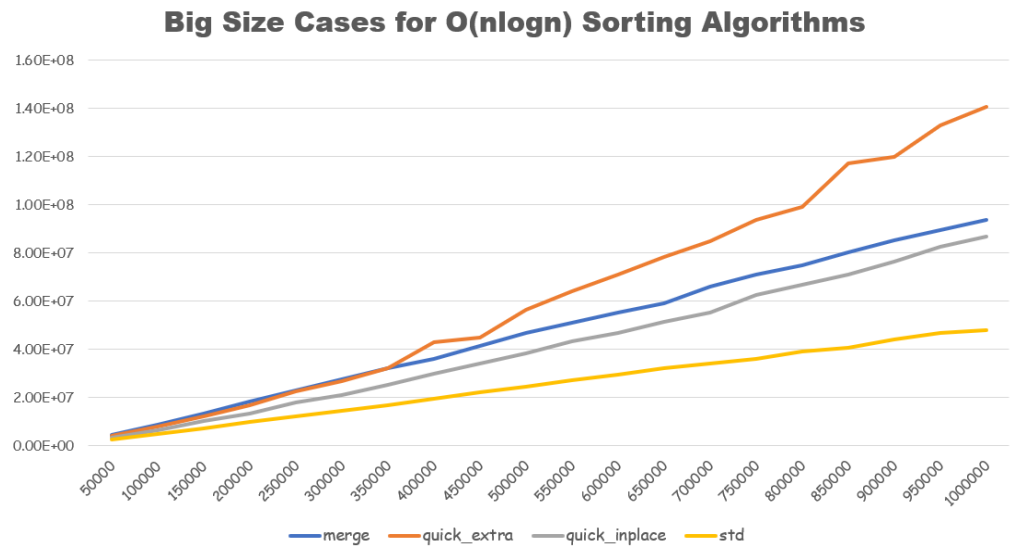

Figure 2: Big Size Cases for $O(nlogn)$ Sorting Algorithms.

We can observe that when the input size becomes bigger, in-place quick sort performs better than extra space quick sort and merge sort, that's because in-place quick sort save the time of allocating some new arrays, though this step only has $O(1)$ time complexity, when the size is quite big, the constant will be large. Also, some in-place operations can be done in cache, while extra space operations must be done in the memory, which costs more time. The `std::sort()` always has the best performance, we will explain it in later part.

Finally, we plot the performance of all sorting algorithms when the size of input arrays is quite small, namely less than 50. The result is calculated by taking average on 5 different input arrays with identical size.

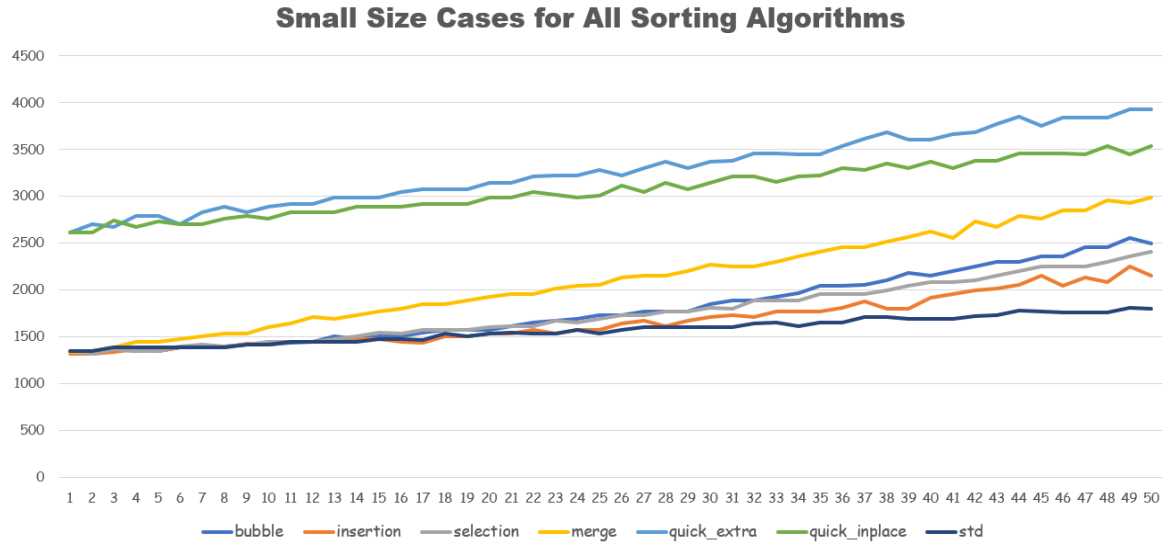

Figure 3: Small Size Cases for All Sorting Algorithms.

We can observe that two $O(nlogn)$ sorting algorithms perform worse when the input size is quite small since they have bigger constant. When the input size is small, there is no big difference between $O(nlogn)$ and $O(n^2)$. The insertion sort performs best, which confirms our expectation.

3 Implementation of std :: sort ()

std :: sort () is not simply quick sort. It's named as intro sort, which is a combination of quick sort, insertion sort and heap sort. We know that insertion sort performs best when the input size is small, so when incoming array has small size (in gcc, this threshold is 16), it will apply insertion sort instead of partition it again. Besides, when the recursion depth is too large, it will apply heap sort, which has $O(nlogn)$ time complexity and $O(1)$ space complexity, to avoid stack overflow. For the quick sort part, it also do some optimization like tail recursion and fast 3-way partitioning. These features make it become the best sorting algorithm in most cases.

4 Reference

1. https://en.cppreference.com/w/cpp/algorithm/sort
2. https://github.com/gcc-mirror/gcc/